Modèle client-serveur

Michel RIVEILL
riveill@unice.fr

Polytech'Nice - Sophia

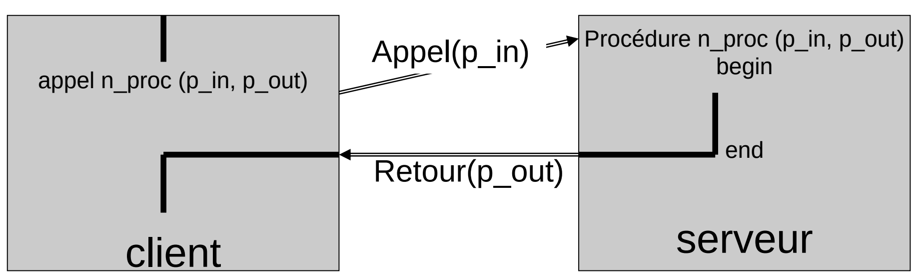
Modèle client-serveur Plan

Modèle client-serveur définition

- application client/serveur
  - application qui fait appel à des services distants au travers d'un échange de messages (les requêtes) plutôt que par un partage de données (mémoire ou fichiers)
  - serveur
    - programme offrant un service sur un réseau (par extension, machine offrant un service)
  - client
    - programme qui émet des requêtes (ou demandes de service). Il est toujours l'initiateur du dialogue

Modèle client-serveur communication par messages

- Deux messages (au moins) échangés
  - Le premier message correspondant à la requête est celui de l'appel de procédure, porteur des paramètres d'appel.
  - Le second message correspondant à la réponse est celui du retour de procédure porteur des paramètres résultats.



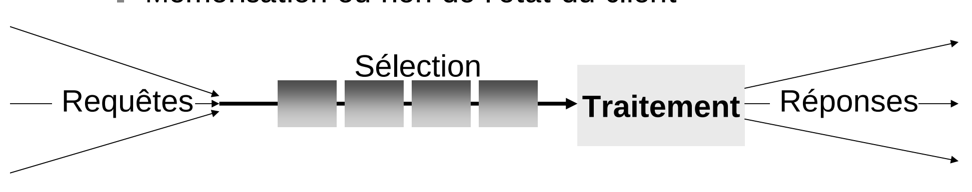
Modèle client-serveur principe

- Vu du client

client — Requête → Service distant

← Réponse

- Vu du serveur
  - Gestion des requêtes (priorité)
  - Exécution du service (séquentiel, concurrent)
  - Mémorisation ou non de l'état du client



Modèle client-serveur exemple

- Serveur de fichiers (aufs, nfsd)
- Serveur d'impressions (lpd)
- Serveur de calcul
- Serveur base de données
- **Serveur de noms (annuaire des services)**

Modèle client-serveur gestion des processus

- Client et serveur sont dans des processus distincts
  - Le client est suspendu lors de l'exécution de la requête
  - Eventuellement, exécution concurrente de plusieurs requêtes chez le serveur
    - Plusieurs processus
      (une mémoire virtuelle associée à chaque processus)
    - Plusieurs processus légers (thread) dans le même espace virtuel
      (contexte restreint : pile, mot d'état, registres)

Mise en œuvre serveur unique

- Processus serveur unique

```
while (true) {
    receive (client_id, message);
    extract (message, service_id, paramètres);
    do_service [service_id] (paramètres, résultats);
    send (client_id, résultats);
};
```

requêtes — **Sélection** — Traitement

réponse

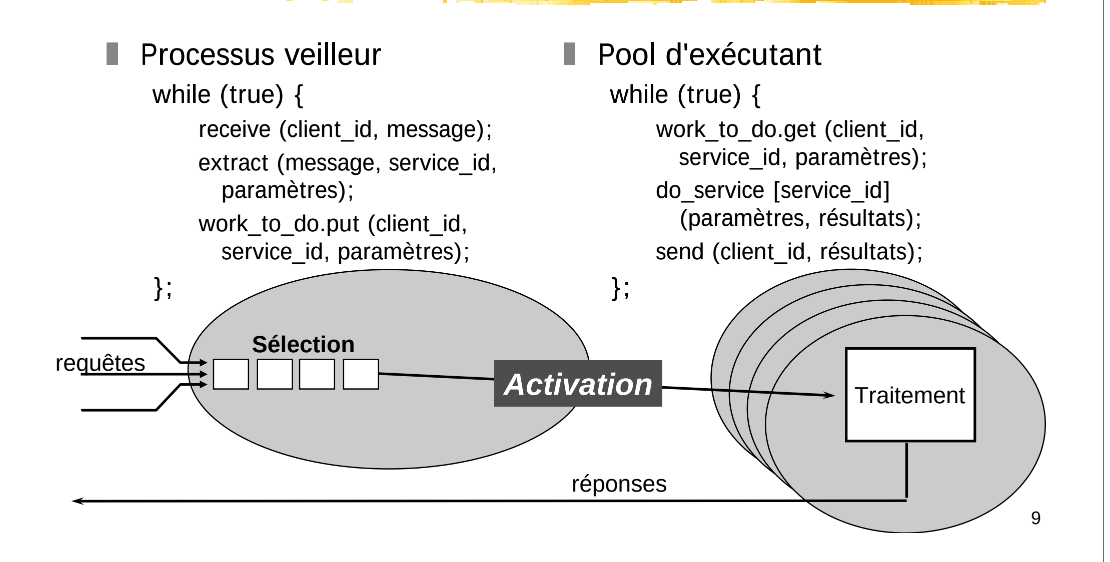
Mise en œuvre
1 processus par service

- Processus veilleur

```
while (true) {
    receive (client_id, message);
    extract (message, service_id,
      paramètres);
    work_to_do.put (client_id,
      service_id, paramètres);
};
```

- Pool d'exécutant

```
while (true) {
    work_to_do.get (client_id,
      service_id, paramètres);
    do_service [service_id]
      (paramètres, résultats);
    send (client_id, résultats);
};
```



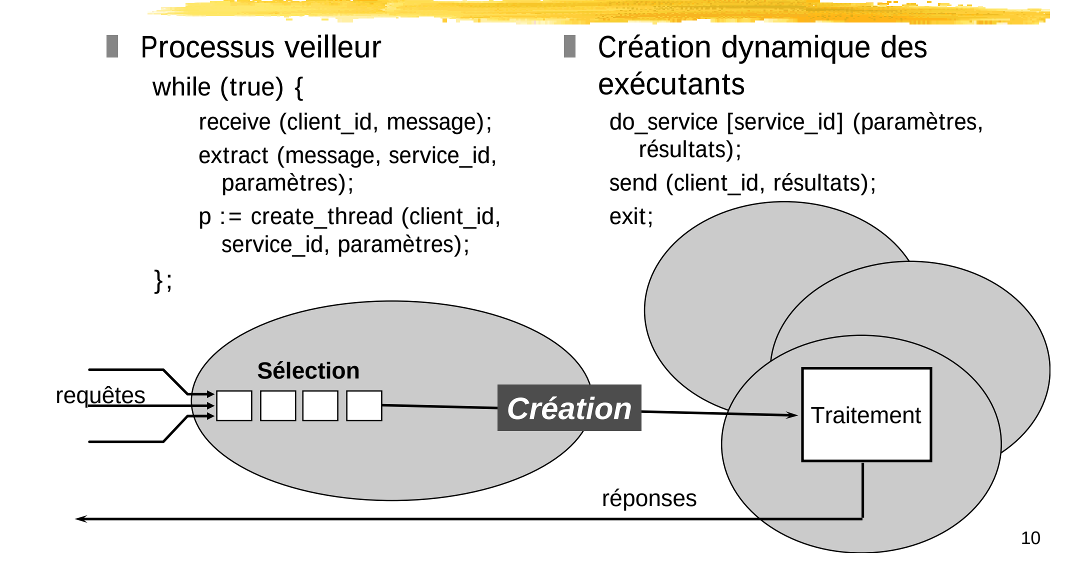
Mise en œuvre
1 processus par service

- Processus veilleur

```
while (true) {
    receive (client_id, message);
    extract (message, service_id,
      paramètres);
    p := create_thread (client_id,
      service_id, paramètres);
};
```

- Création dynamique des exécutants

```
do_service [service_id] (paramètres,
  résultats);
send (client_id, résultats);
exit;
```



Différents types de service
pas de donnée rémanente (ou persistante)

- situation idéale ou le service s 'exécute uniquement en fonction des paramètres d 'entrée
  - pas de modification de données rémanente sur le serveur
- solution très favorable
  - pour la tolérance aux pannes
  - pour le contrôle de la concurrence
- exemple :
  - calcul d 'une fonction scientifique

Différents types de service
avec donnée rémanente (ou persistante)

- Les exécutions successives manipulent des données persistantes
  - modification du contexte d 'exécution sur le site distant
  - problèmes de contrôle de la concurrence
  - difficultés en cas de panne en cours d 'exécution
- Exemples
  - Etat d 'un objet manipulé par ses méthodes
  - Serveur de fichier réparti
    - Pose de verrou pour les opérations d 'écriture

Différents types de service
mode sans état

- Les appels de procédure s ’exécutent sans lien entre eux
  - il peut y avoir modification de données globales mais l ’opération s ’effectue sans lien avec celles qui l ’ont précédé
- exemple
  - serveur de fichiers répartis : accès aléatoire
    - écriture du $n^{ième}$ article d ’un fichier
    - NFS (Network File System de SUN - SGF réparti

Différents types de service
mode avec état

- Les appels successifs s ’exécutent en fonction d ’un état laissé par les appels antérieurs
  - gestion de l ’ordre des requêtes est indispensable
- exemple
  - serveur de fichiers répartis : accès séquentiel
  - appel de méthode sur un objet

Appel de procédure à distance (RPC)

- Outils de base pour réaliser le mode client-serveur.
  - L'opération à réaliser est présentée sous la forme d'une procédure que le client peut faire exécuter à distance par un autre site : le serveur.
    - Forme et effet identique à ceux d'un appel local
      - Simplicité (en l'absence de pannes)
      - Sémantique identique à celle de l'appel local
  - Opérations de base
    - Client
      - **doOp** (IN Port serverId, Name opName, Msg *arg, OUT Msg *result)
    - Serveur
      - **getRequest** (OUT Port clientId, Message *callMessage)
      - **sendReply** (IN Port clientId, Message *replyMessage)
      - **opName** (Msg *arg, OUT Msg *result)

RPC
Objectifs

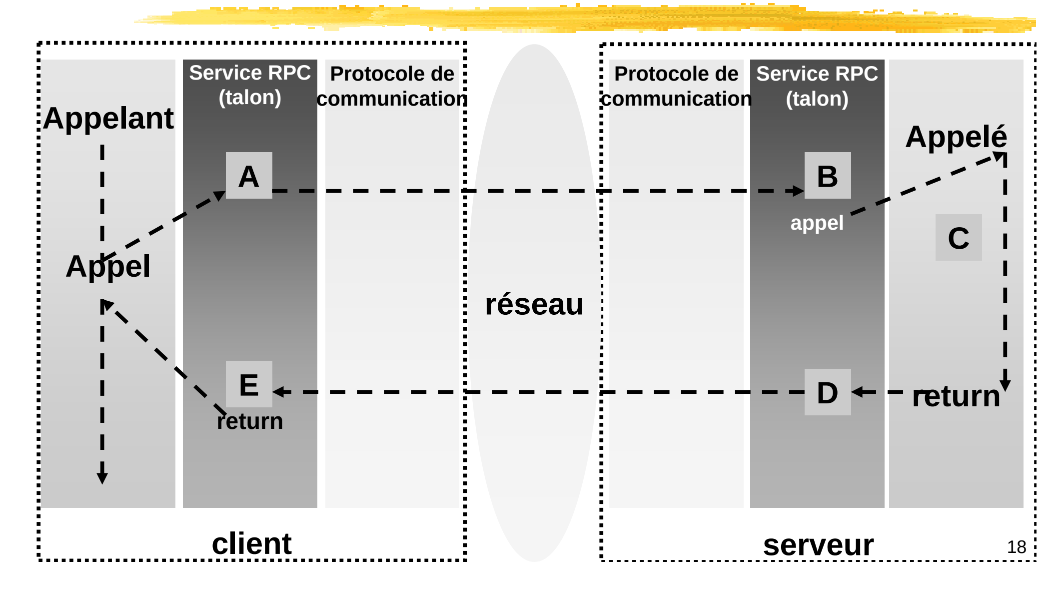
- Retrouver la sémantique habituelle de l ’appel de procédure
  - sans se préoccuper de la localisation de la procédure
  - sans se préoccuper du traitement des défaillances
- Objectifs très difficiles à atteindre
  - réalisation peu conviviale
  - sémantique différente de l ’appel de procédure même en l ’absence de panne

RPC : les pièges

- Appel de procédure
- appelant et appelé sont dans me même espace virtuel
  - même mode de pannes
  - appel et retour de procédure sont des mécanismes internes considérés comme fiables
    - sauf aspect liés à la liaison dynamique de la procédure et à la vérification de la protection
  - dans certains langages
    - mécanisme d'exception pour transmettre les erreurs de l'appelé à l'appelant

- Appel de procédure à distance
- Appelant et appelé sont dans 2 espaces virtuels différents
  - pannes du client et du serveur sont indépendantes
  - pannes du réseau de communication (perte du message d'appel ou de réponse)
  - temps de réponse du serveur long
    - charge du réseau ou du site serveur

RPC [Birrel & Nelson 84] Principe de réalisation



RPC (A) Principe de fonctionnement

- Côté de l 'appelant
  - Le client réalise un appel procédural vers la procédure talon client
    - transmission de l 'ensemble des arguments
  - au point A
    - le talon collecte les arguments et les assembles dans un message (empaquetage - parameter marshalling)
    - un identificateur est généré pour le RPC
    - Un délai de garde est armé
    - Pb : détermination de l 'adresse du serveur
    - le talon transmet les données au protocole de transport pour émission sur le réseau

RPC (B et C) Principe de fonctionnement

- Coté de l 'appelé
  - le protocole de transport délivre le message au service de RPC (talon serveur/skeleton)
  - au point B
    - le talon désassemble les arguments (dépaquetage - unmarshalling)
    - l 'identificateur de RPC est enregistré
  - l 'appel est ensuite transmis à la procédure distante requise pour être exécuté (point C). Le retour de la procédure redonne la main au service de RPC et lui transmet les paramètres résultats (point D)

RPC (D)
Principe de fonctionnement

- Coté de l 'appelé
  - au point D
    - les arguments de retour sont empaquetés dans un message
    - un autre délai de garde est armé
    - le talon transmet les données au protocole de transport pour émission sur le réseau

RPC (E)
Principe de fonctionnement

- Coté de l 'appelant
  - l 'appel est transmis au service de RPC (point E)
    - les arguments de retour sont dépaquetés
    - le délai de garde armé au point A est désarmé
    - un message d'acquittement avec l'identificateur du RPC est envoyé au talon serveur (le délai de garde armé au point D peut être désarmé)
    - les résultats sont transmis à l 'appelant lors du retour de procédure

RPC
Rôle des talons

Talon client - stub

- C'est la procédure d'interface du site client
  - qui reçoit l'appel en mode local
  - le transforme en appel distant en envoyant un message.
  - reçoit les résultats après l'exécution
  - retourne les paramètres résultats comme dans un retour de procédure.

Talon serveur - skeleton

- C'est la procédure sur le site serveur
  - qui reçoit l'appel sous forme de message,
  - fait réaliser l'exécution sur le site serveur par la procédure serveur (choix de la procédure)
  - retransmet les résultats par message.

RPC
Problèmes

- Traitement des défaillances
  - Congestion du réseau ou du serveur
    - la réponse ne parvient pas avant une date fixée par le client (système temps critique)
  - Panne du client pendant le traitement de la requête
  - Panne du serveur avant ou pendant le traitement de la requête
  - Erreur de communication
- Problèmes de sécurité
  - authentification du client
  - authentification du serveur
  - confidentialité des échanges Performance
- Désignation et liaison,
- Aspects pratiques
  - Adaptation à des conditions multiples (protocoles, langages, matériels)
  - Gestion de l'hétérogénéité

Différents types de pannes

- Panne du serveur
  - attente indéfinie par le client d'une réponse qui ne viendra peut être jamais
    - utilisation d'une horloge de garde
    - le client décide de la stratégie de reprise
      - abandon
      - re-essaie
      - choix d'un autre serveur
    - risque d'exécuter plusieurs fois la même procédure

Différents types de pannes

- Panne du client
  - risque de réalisation de travaux inutiles
  - risque de confusion après relance du client entre les nouvelles réponses attendues et les réponses de l'ancienne requête
  - le serveur est dit orphelin
    - nécessité de détruire les tâches serveurs orphelines et de distinguer les requêtes utiles des vieilles requêtes
  - Pbs : l'état du client et du serveur peuvent devenir inconsistants

RPC
Sémantique en cas d 'erreur

- En cas d'erreur : la sémantique est définie par le mécanisme de reprise
  - Indéfini
  - Au moins une fois
    - Plusieurs appels possibles si perte de la réponse
    - Acceptable si opération idempotente (f o f = f)
  - Au plus une fois
    - si expiration du délai A alors code d 'erreur sinon résultat correct
    - pas de mécanisme de reprise
  - Exactement une fois (idéal...)
    - si expiration du délai A alors re-émission de l 'appel
    - si expiration du délai D alors re-émission du résultat

RPC
Traitement des défaillances

- Congestion du réseau ou du serveur
  - Panne transitoire (ne nécessite pas d 'intervention)
  - Détection : expiration du délai de garde A ou D
  - Recouvrement
    - le service de RPC (point A) re-émet l 'appel (même id) sans intervention de l 'application
    - Le service de RPC (point C) détecte qu'il s 'agit d 'une re-émission
      - si l 'appel est en cours d 'exécution : aucun effet
      - si le retour a déjà été effectué, re-émission du résultat
  - Sémantique : Exactement UN
    - (en l 'absence de panne permanente du client, du serveur ou du réseau)

RPC
Traitement des défaillances

- Panne du client après émission de l 'appel
  - L 'appel est correctement traité
    - changement d 'état du serveur
    - l 'appel de procédure est déclaré orphelin
  - Détection : expiration du délai de garde D
  - Recouvrement
    - L 'application cliente re-émet l 'appel (avec id différent)
      - sémantique : Au moins UN
    - le serveur ne peut pas détecté qu 'il s 'agit d 'une répétition
      - service idempotent : pas d 'incidence
      - service non idempotent
        - service transactionnel (annulation par le client des effets de l 'appel orphelin)

RPC
Traitement des défaillances

- Panne du serveur après émission de l 'appel
  - L 'appel peut être correctement ou partiellement traité
    - avant point B, durant C ou avant point D
  - Détection : expiration du délai de garde A
  - Recouvrement
    - le client doit re-émettre l 'appel dès que le serveur redémarre
      - sémantique : Au moins UN
      - le client ne connaît pas l 'endroit de la panne
        - si avant point B : pas d 'incidence
        - si entre B et D : changement d 'état du serveur
      - service transactionnel pour mémoriser id et état avant exécution

RPC
Représentation des données

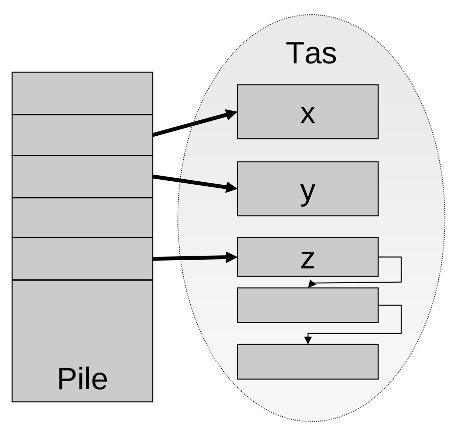
- Problème classique dans les réseaux
  - Conversion est nécessaire si le site client et le site serveur
    - n 'utilise pas le même codage (big endian, little endian)
    - utilisent des formats internes différents (type caractère, entier, flottant, ...)
    - solution placée classiquement dans la couche 6 du modèle OSI présentation
  - dans réseau : passage de paramètres par valeur
    - émulation des autres modes

RPC
Représentation des données

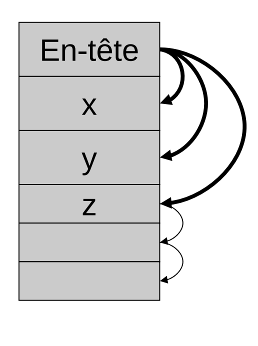
- solution normalisée :
  - syntaxe abstraite de transfert ASN 1
- autres solutions
  - Représentation externe commune : XDR Sun (non optimal si même représentation)
  - Représentation locale pour le client, conversion par le serveur
  - Choix d'une représentation parmi n (standard), conversion par le serveur
  - Négociation client serveur

RPC
Passage des paramètres

- Format du paquet d 'appel
  - Appel local



- Appel distant



Passage des paramètres
sémantiques de transmission variées

- Valeur
  - pas de problème particulier
- copie/restauration
  - valeurs des paramètres sont recopiées
  - pas de difficultés majeures mais redéfinition des solutions définies pour les réseaux
  - optimisation des solutions pour le RPC
  - bonne adaptation au langage C (faiblement typé)

Passage des paramètres
sémantiques de transmission variées

- référence
  - utilise une adresse mémoire centrale du site de l 'appelant... aucun sens pour l 'appelé
  - solutions
    - interdiction totale
      - introduit une différence entre procédures locales et procédures distantes
    - simulation en utilisation une copie de restauration
      - marche dans de très nombreux cas
      - mais violation dans certains cas de la sémantique du passage par référence
      - exemple de pb : procédure double_incr (x, y)

```
                                   x := x+1;
                                   y := y+1;
a := 0 ; double_incr (a, a)
résultat : a = 2 ou a = 1 ?
```


Passage des paramètres
sémantiques de transmission variées

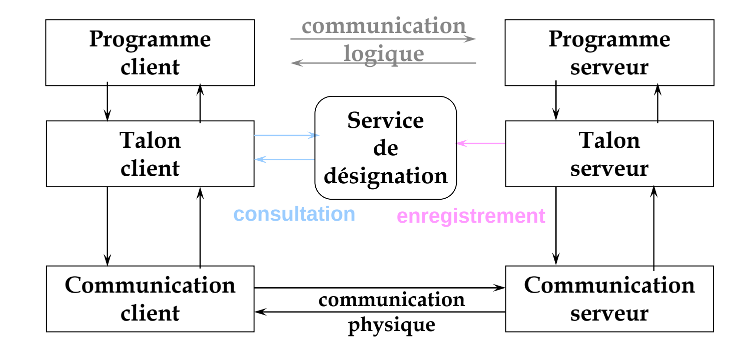
- Référence
  - solutions (suite)
    - reconstruire l 'état de la mémoire du client sur le site serveur
      - solutions très coûteuse
    - utilisation d 'une mémoire virtuelle répartie
      - nécessite un système réparti avec mémoire virtuelle
- Solutions généralement prises
  - IN : passage par valeur (aller)
  - OUT : passage par valeur (retour)
  - IN-OUT : interdit ou passage par copie-restauration
    - ATTENTION : n 'est pas équivalent au passage par référence

RPC
Désignation

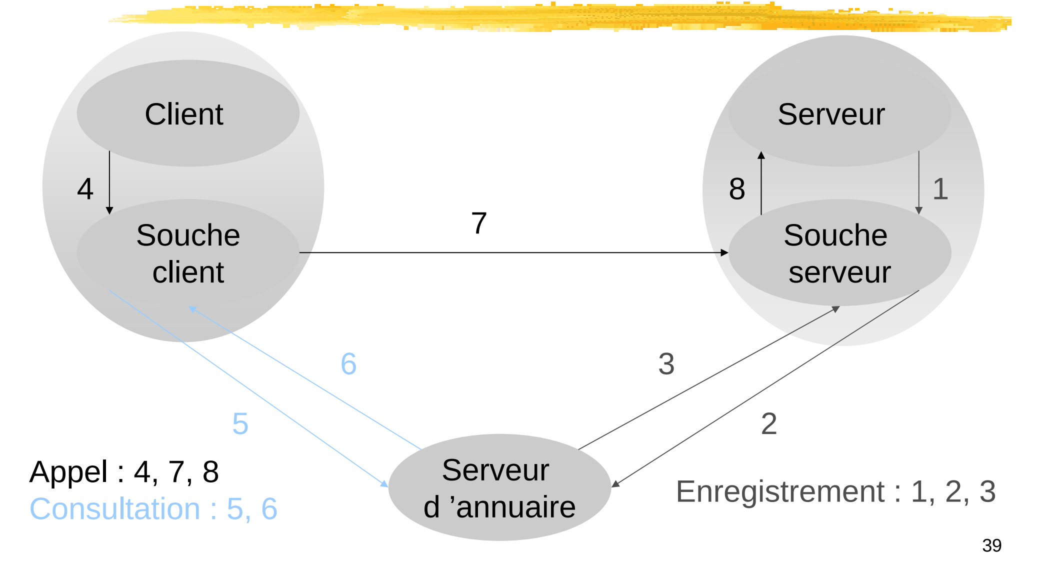
- Objets à désigner
  - Le site d 'exécution, le serveur, la procédure
  - Désignation globale indépendante de la localisation
    - possibilité de reconfiguration des services (pannes, régulation de charge, ...)
- Désignation
  - statique ou dynamique
    - statique : localisation du serveur est connue à la compilation
    - dynamique : non connue à la compilation, objectifs :
      - séparer connaissance du nom du service de la sélection de la procédure qui va l 'exécuter
      - permettre l 'implémentation retardée

Liaison et fonctionnement

- Liaison (détermination de l'adresse du serveur)
  - Liaison statique (pas d 'appel à un serveur de nom ou appel lors de la compilation)
  - Liaison au premier appel (appel du serveur de nom lors du premier appel)
  - Liaison à chaque appel (appel du serveur de nom à chaque appel)



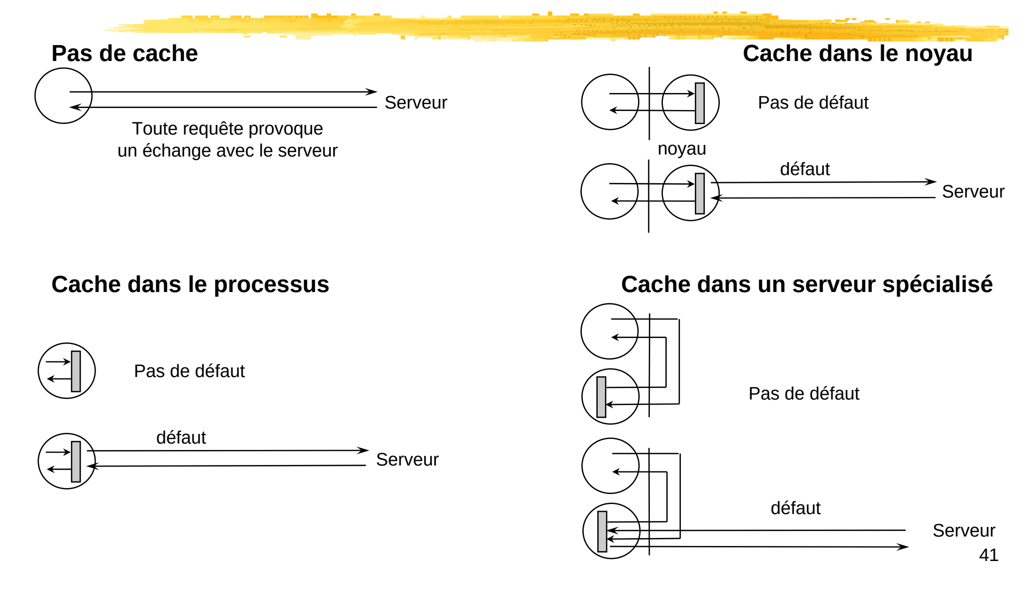
Liaison - solution classique
DNS Internet



Liaison - solution classique
DNS Internet

- Etape 1, 2 et 3
  - enregistrement dans une base de données des noms de serveurs
    - étape 2 : transmission du nom du service, adresse réseau du service, tous attributs complémentaires nécessaire (si plusieurs serveurs rendent le même service)
    - étape 3 : enregistrement confirmé (pas d homonymie, etc.)
- Etape 5 et 6
  - liaison entre un client et un serveur
- Etape 4, 7, 8, ...
  - réalisation de l 'appel de procédure souhaité

RPC
Performance : utilisation de cache



Validation du contenu du cache client

- A la charge du client
  - le client interroge le serveur pour savoir si sa copie est toujours valide
  - vérification par comparaison d'une date associée à la dernière modification des données
  - comparaison périodique ? à chaque accès ? à chaque ouverture ?
- A la charge du serveur
  - la copie sur le serveur est la copie de référence
  - le serveur possède la liste des clients qui possèdent une copie
  - le serveur prévient chaque client de la modification de la copie maître (mécanisme de call-back)
  - **TRES LOURD**
  - Remarque : extension du modèle client-serveur puisque le serveur prend l'initiative de l'échange

RPC
Mise en œuvre

- Directement
  - Le programmeur se charge de tout...
    - socket Unix
- Utilisation d'un langage de description d'interface
  - Le programmeur spécifie l 'interface des services accessibles à distance
    - Rpcgen
    - Java RMI (RPC Objet)
- Intégration dans un langage de programmation
  - Approche totalement transparente (tous les objets peuvent éventuellement être accédé à distance):
    - langage Guide (RPC Objet)

Modèle client-serveur
Mise en œuvre : envoi de message

- Mise en œuvre à l 'aide d 'un mécanisme d 'envoi de message
  - par exemple : socket UNIX

Mise en œuvre du client/serveur à l'aide des sockets

- Point d'accès à différents services de communication
  - avec ou sans connexion
  - différentes familles de protocoles (ISO, Internet, Xerox NS, etc...)
- Socket créée dynamiquement par un processus
  - s = **socket** (PF_UNIX/PF_INET, SOCK_STREAM/SOCK_DGRAM, 0)
  - A la création on précise
    - la famille de protocoles (par exemple Unix, Inet)
    - le type de service (par exemple : stream ou datagram)
    - éventuellement l'identification du protocole choisi,
- Une fois créé une socket doit être liée à un point d'accès
  - ret = **bind** (s, monAdresse, longueurDe_monAdresse)

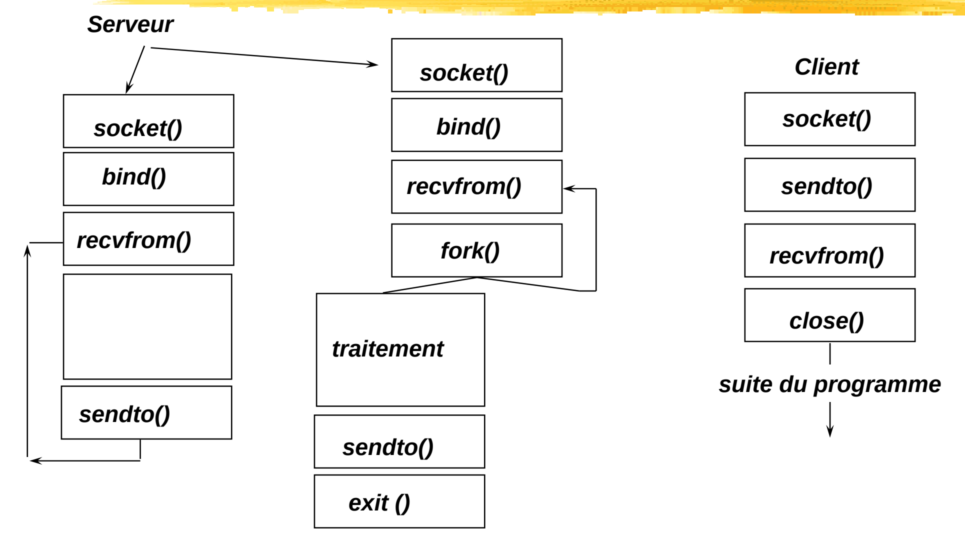
Algorithme d'un serveur itératif en mode non connecté

- Dans ce mode le client peut envoyer des appel au serveur à n 'importe quel moment
  - mode assez léger orienté
    - traitement non ordonné des appels
    - absence de mémoire entre appels successifs (serveur sans données rémanentes et sans état)
  - exemple :
    - calcul de fonction numérique
    - DNS
    - NFS

Utilisation du mode non connecté

- caractéristiques
  - pas d'établissement préalable d'une connexion
  - adapté aux applications pour lesquelles les réponses aux requêtes des clients sont courtes (un message)
  - protocole de transport utilisé : UDP
  - mode d'échange par messages : le récepteur reçoit les données suivant le même découpage que celui effectué par l'émetteur
- contraintes
  - le client doit avoir accès à l'adresse du serveur (adresse IP et numéro de port)
  - pour répondre à chaque client, le serveur doit en récupérer l'adresse : il faut pour cela utiliser les primitives sendto et recvfrom
- mode de gestion des requêtes
  - itératif : le processus serveur traite les requêtes les unes après les autres

Enchaînement des opérations en mode non connecté



Algorithme d'un serveur itératif en mode connecté

- Le client
  - ouvre une connexion avec le serveur avant de pouvoir lui adresser des appels, puis ferme la connexion à la fin de la suite d 'opération
    - délimitation temporelle des échanges
    - maintien de l 'état de connexion pour la gestion des paramètres de qualité de service
      - traitement des pannes, propriété d 'ordre
  - orienté vers
    - traitement ordonné d 'une suite d 'appel
      - ordre local (requête d 'un client traitée dans leur ordre d 'émission), global ou causal
    - la gestion de données persistantes ou de protocole avec état

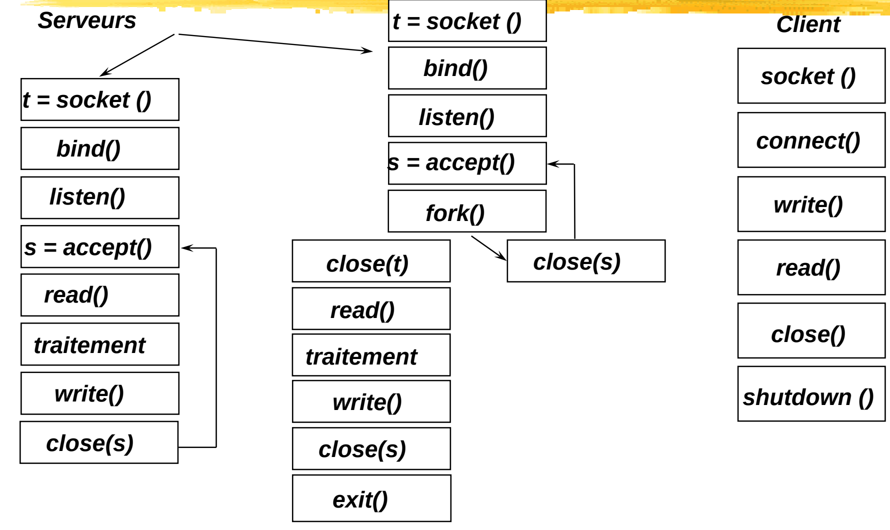
Utilisation du mode connecté

- caractéristiques
  - établissement préalable d'une connexion (circuit virtuel) : le client demande au serveur s'il accepte la connexion
  - fiabilité assurée par le protocole de transport utilisé : TCP
  - mode d'échange par flots d'octets : le récepteur n'a pas connaissance du découpage des données effectué par l'émetteur
  - possibilité d'émettre et de recevoir des caractères urgents (OOB : Out Of Band)
  - après initialisation, le serveur est "passif", il est activé lors de l'arrivée d'une demande de connexion d'un client
  - un serveur peut répondre aux demandes de services de plusieurs clients : les requêtes arrivées et non traitées sont stockées dans une file d'attente

Utilisation du mode connecté

- contrainte
  - le client doit avoir accès à l'adresse du serveur (adresse IP et numéro de port)
- modes de gestion des requêtes
  - itératif : le processus serveur traite les requêtes les unes après les autres
  - concurrent : par création de processus fils pour les échanges de chaque requête

Enchaînement des opérations en mode connecté



Mise en œuvre directe

- Permet de comprendre les mécanismes qui sont utilisés par les autres approches
- A proscrire, car pour chaque application, le programmeur traite les mêmes problèmes
  - Génération « à la main des talons / squelette »
    - empaquettage / dépaquettage des paramètres
    - désignation et liaison du serveur
    - traitement de l 'hétérogénéité
    - traitement des défaillances

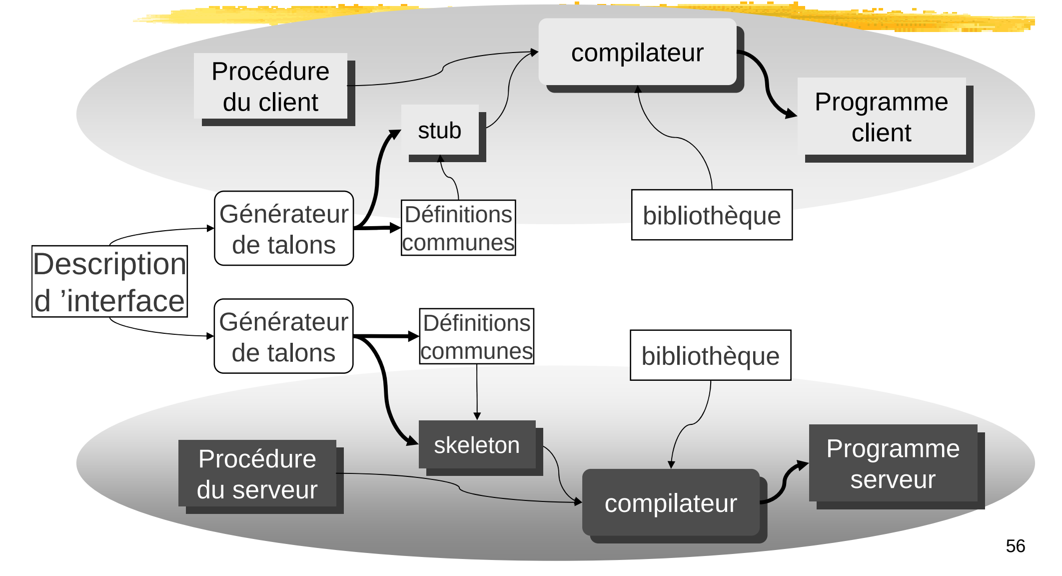
RPC
Intégration dans un langage

- Objectifs : faciliter l 'écriture du client et du serveur
  - Approche non transparente
    - l 'appel distant est syntaxiquement différente d 'un appel local
      - a_type := call a_remote_proc(<args>) <one, maybe> timeout N
  - Approche transparente
    - détection automatique de l 'appel distant
      - comment ?
        - liste de noms ou lors de la liaison
        - **du coté du client :** l 'appel à la procédure est remplacé par un appel au talon client (stub) généré par le compilateur
        - **du coté du serveur :** le compilateur produit un talon serveur (skeleton) capable de recevoir l 'appel et de le rediriger vers la procédure
  - Un outil : les langages de définition d 'interface (IDL)

RPC
IDL : spécification des interfaces

- Utilisation d'un langage
  - Spécification commune au client et au serveur adaptée à des langages multiples
    - analogie avec les modules d'Ada, Modula-2
  - Définition des types et natures des paramètres (IN, OUT, IN-OUT)
- Utilisation de ces définitions pour générer automatiquement :
  - le talon client (ou proxy, stub)
  - le talon serveur (ou squelette, skeleton)

IDL
Mode opératoire



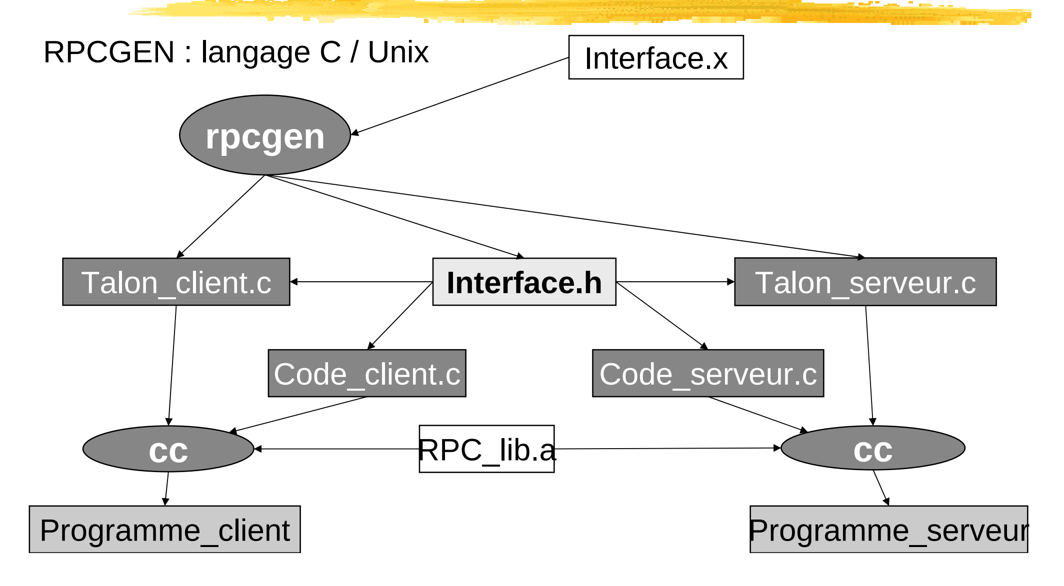
RPC
Compilation des interfaces

- Produire les talons client et serveur
  - différents langages cibles
  - Talons client et serveur générés avec la même version du compilateur et de la spécification
    - vérification par estampille
- Procédure générée
  - empaquetage des paramètres
  - identification de la procédure à appeler
  - procédure de reprise après expiration des délais de garde
- Coté client
  - Remplacer les appels de procédure distants par des appels au talon client
- Coté serveur
  - Au démarrage le serveur se fait connaître (enregistrement dans un service de désignation)
  - recevoir l 'appel et affectué l 'appel sur la procédure

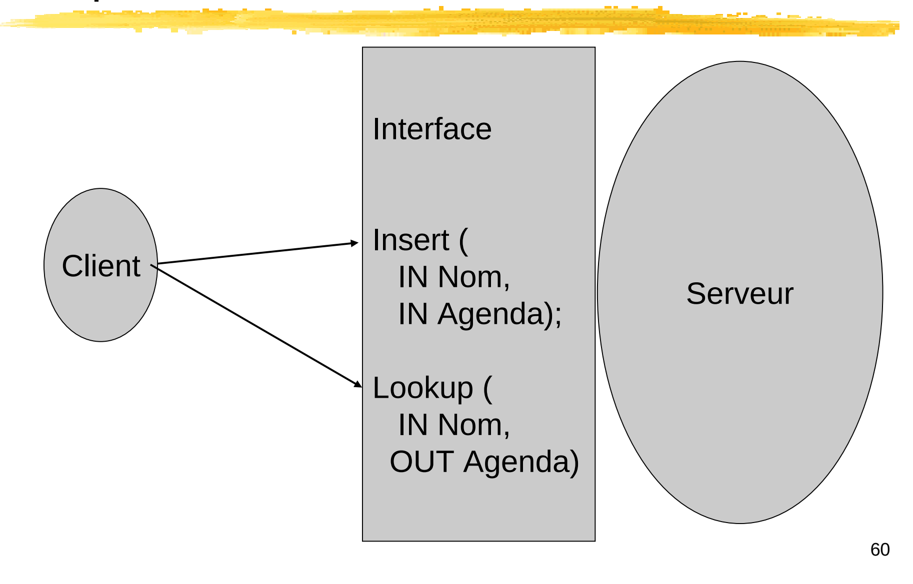
RPC
Avantage d 'un IDL

- Traitement de l'hétérogénéité
- Types de données différents selon les langages
  - Représentations internes différentes selon les systèmes
  - Définition des types indépendante de la représentation
- Description d'une interface
  - Description des types élémentaires (arguments, résultats, exceptions)
  - Noms des procédures de conversion pour types complexes (avec pointeurs)

IDL
exemple de mise en œuvre



RPCGEN
exemple d 'utilisation



Rpcgen : interface.x description de l'interface

```
const MAX_NAME = 255;

typedef char Name <MAX_NAME>;
typedef ....... Agenda;
typedef long status;
struct entry {Name name; Agenda agenda;}; typedef struct entry Entry;

program DISTR_AGENDA {
   version VERSION_NUMBER {
      Agenda Lookup (Name) = 1;          // numéro de la procédure
      status Insert (Entry) = 2;         // numéro de la procédure
   } = 1                    // numéro de version
} = 76                      // numéro de programme
```

- **rpcgen interface.x**
  - produit les fichiers : interface.h, interface_xdr.c, interface_svc.c et interface_clnt.c

Rpcgen : programmation du serveur

rpcgen -Ss interface.x >serveur.c

```
/* This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */
#include "interface.h"

Agenda *lookup_1(argp, rqstp)
    Name *argp; struct svc_req *rqstp;
{
    static Agenda  result;
    /* insert server code here */
    printf ( "serveur : lookup %s\n",
            argp->Name_val);

    return (&result);
}
```

```
status *insert_1(argp, rqstp)
    Entry *argp;
    struct svc_req *rqstp;
{
    static status  result;

    /* insert server code here */
    printf ( "serveur : insert %s-%d\n",
            argp->name.Name_val,
            argp->agenda);

    return (&result);
}
```


Rpcgen : programmation du client

rpcgen -Sc interface.x >client.c

```
/* This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */
#include "interface.h"

void distr_agenda_1(host)
    char *host;
{
    CLIENT *clnt;
    Agenda  *result_1;
    Name  lookup_1_arg;
    status  *result_2;
    Entry  insert_1_arg;
// page suivante...
}
```

```
main(argc, argv)
    int argc;
    char *argv[];
{
    char *host;

    if (argc < 2) {
        printf("usage:  %s server_host\n",
argv[0]);
        exit(1);
    }
    host = argv[1];
    distr_agenda_1(host);
}
```


Rpcgen : programmation du client

```
#ifndef    DEBUG
    clnt = clnt_create(host, DISTR_AGENDA,
    VERSION_NUMBER, "netpath");
    if (clnt == (CLIENT *) NULL) {
        clnt_pcreateerror(host);
        exit(1);
    }
#endif     /* DEBUG */

lookup_1_arg.Name_len=strlen("Michel")+1;
lookup_1_arg.Name_val="Michel";

    result_1 = lookup_1(&lookup_1_arg, clnt);
    if (result_1 == (Agenda *) NULL) {
        clnt_perror(clnt, "call failed");
    }

printf("retour lookup : agenda = %d\n", *result_1);
```

```
insert_1_arg.name.Name_len=strlen("Michel")+1;
insert_1_arg.name.Name_val="Michel";
insert_1_arg.agenda=12;

    result_2 = insert_1(&insert_1_arg, clnt);
    if (result_2 == (status *) NULL) {
        clnt_perror(clnt, "call failed");
    }

printf("retour insert : status = %d\n", *result_2);
```

Rpcgen : construction du makefile

- rpcgen -Sm interface.x > Makefile
- Modifier les lignes :
  - SOURCES_CLNT.c = client.c
  - SOURCES_SVC.c = serveur.c
  - CLIENT = client
  - SERVER = serveur
- make
  - produit deux binaires : client et serveur

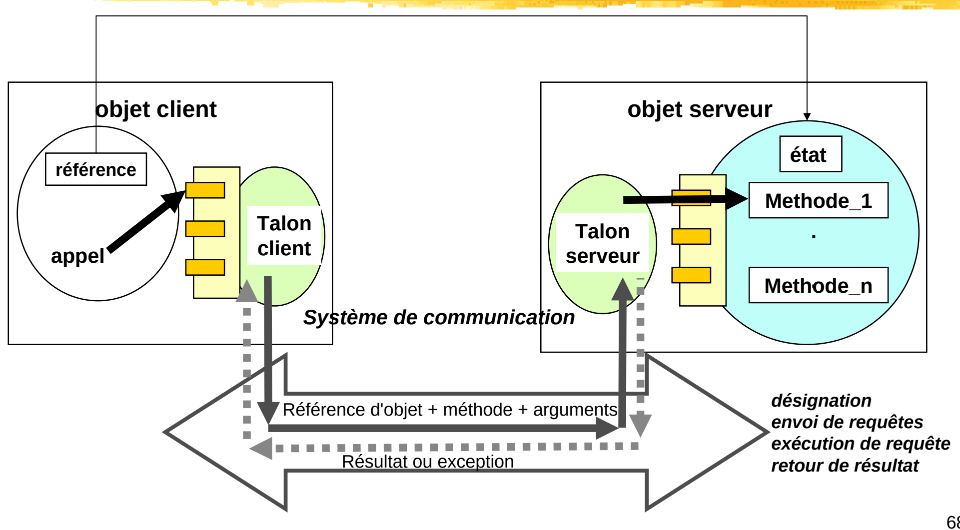
Sun RPC : limitations

- Avec d'UDP : taille d 'un message < 8 K octets.
- Un seul paramètre d 'appel et un seul de retour
  - si plusieurs paramètres construire une structure
  - formatage des paramètres complexes à l 'aide de XDR
- Sémantique en cas de panne : au moins un
  - réémission jusqu'à la réponse ou erreur
- Aucune facilité pour écrire un serveur multiplexé.

Client-serveur « à objet »

- Motivations
  - propriétés de l'objet (encapsulation, modularité, réutilisation, polymorphisme, composition)
  - objet : unité de désignation et de distribution
- éléments d'une "invocation"
  - référence d'objet ("pointeur" universel)
  - identification d'une méthode
  - paramètres d'appel et de retour (y compris signal d'exception)
    - passage par valeur : types élémentaires et types construits
    - passage par référence
- **objets "langage"**
  - représentation propre au langage : instance d'une classe
  - exemple : Java RMI
- **objets "système"**
  - représentation "arbitraire" définie par l'environnement d'exécution
  - exemple : CORBA

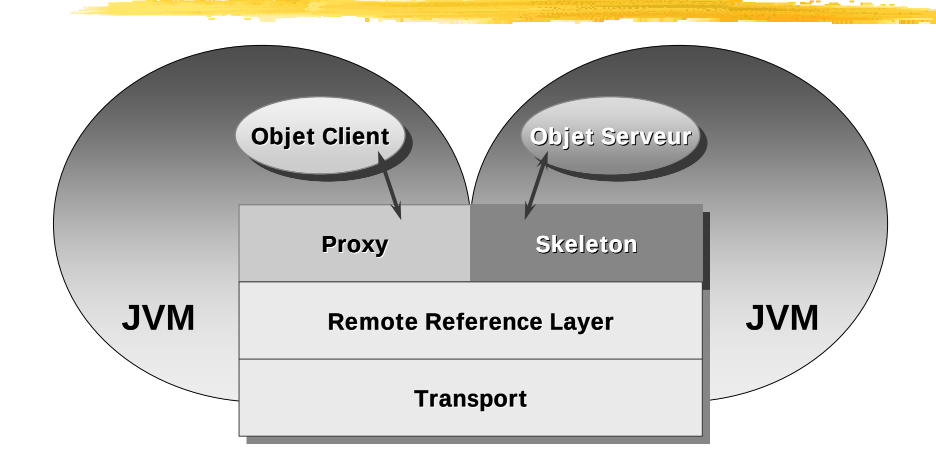
Appel de méthode à distance
Remote Method Invocation (RMI)



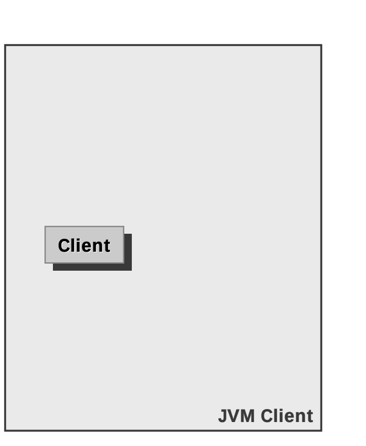
RPC
Java RMI

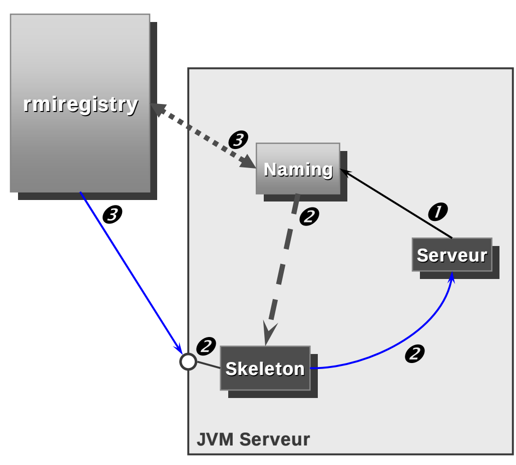
- Un RPC objet intégré à Java
- Interaction d'objets situés dans des espaces d'adressage différents sur des machines distinctes
- Simple à mettre en œuvre : un objet distribué se manipule comme tout autre objet Java

Java RMI
Architecture



Java RMI
Architecture



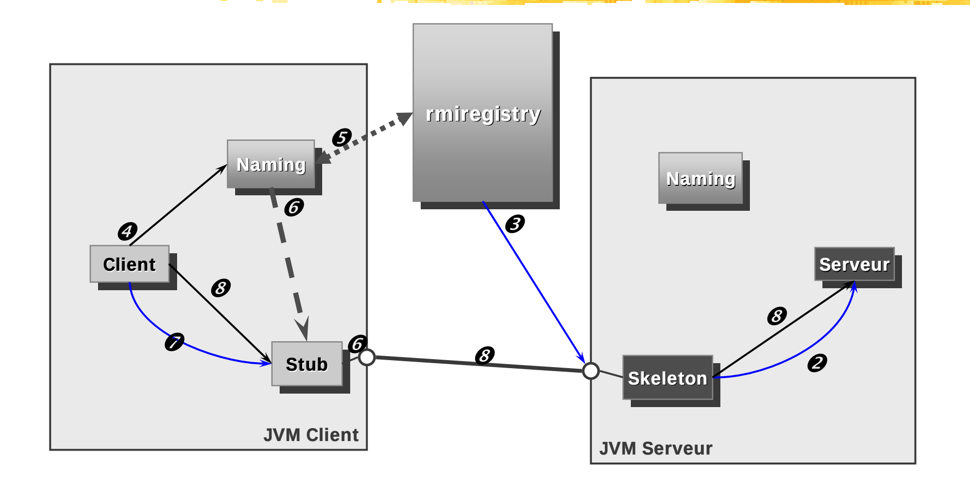


Java RMI
Mode opératoire coté serveur

- 1 - L'objet serveur s'enregistre auprès du Naming de sa JVM (méthode *rebind*)
- 2 - L 'objet skeleton est créé, celui-ci crée le port de communication et maintient une référence vers l'objet serveur
- 3 - Le Naming enregistre l'objet serveur, et le port de communication utilisé auprès du serveur de noms

- L'objet serveur est prêt à répondre à des requêtes

Java RMI
Architecture



Java RMI
Mode opératoire coté client

- 4 - L'objet client fait appel au Naming pour localiser l'objet serveur (méthode lookup)
- 5 - Le Naming récupère les "références" vers l'objet serveur, ...
- 6 - crée l 'objet Stub et ...
- 7 - rend sa référence au client
- 8 - Le client effectue l'appel au serveur par appel à l 'objet Stub

Java RMI
Manuel d'utilisation

- Définition de l'interface de l'objet réparti
  - interface : "extends java.rmi.Remote"
  - methodes : "throws java.rmi.RemoteException »
  - paramètres sérializable : "implements Serializable"
- Ecrire une implémentation de l'objet serveur
  - classe : "extends java.rmi.server.UnicastRemoteObject"

Java RMI
Mode opératoire

- codage
  - description de l 'interface du service
  - écriture du code du serveur qui implante l 'interface
  - écriture du client qui appelle le serveur
- compilation
  - compilation des sources (javac)
  - génération des stub et skeleton (rmic)
- activation
  - lancement du serveur de noms (rmiregistry)
  - lancement du serveur
  - lancement du client

RPC
Java RMI : écriture de l 'interface

- Mêmes principes de base que pour l 'interface d 'un objet local
- Principales différences
  - l 'interface distante doit être publique
  - l 'interface distante doit étendre l 'interface java.rmi.Remote
  - chaque méthode doit déclarer au moins l 'exception java.rmi.RemoteException
  - tout objet distant passé en paramètre doit être déclaré comme une interface (passage de la référence de l 'objet)
  - tout objet local passé en paramètre doit être sérialisable

Java RMI
Exemple : Interface

fichier Hello.java

```
public interface Hello extends java.rmi.Remote {
  String sayHello() throws java.rmi.RemoteException;
}
```

Description de l 'interface

RPC
Java RMI : écriture du serveur

- Serveur = la classe qui implémente l 'interface
  - spécifier les interfaces distantes qui doivent être implémentées
    - objets locaux passés par copie (il doivent implémenter l 'interface java.io.serialisable)
    - objets distants passés par référence (actuellement référence à un stub)
  - c 'est un objet java standard
    - définir le constructeur de l 'objet
    - fournir la mise en œuvre des méthodes pouvant être appelée à distance
    - ainsi que celle des méthodes n 'apparaissant dans aucune interface implémentée
  - créer au moins une instance du serveur
  - enregistrer au moins une instance dans le serveur de nom (rmiregistry)

Java RMI
Exemple : Serveur

fichier HelloServeur.java

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloServeur extends UnicastRemoteObject implements Hello {
  private String msg;

  // Constructeur
  public HelloServeur(String msg) throws java.rmi.RemoteException {
        super();
        this.msg = msg;
    }

  // Implémentation de la méthode distante.
  public String sayHello() throws java.rmi.RemoteException {
        return "Hello world: " + msg;
    }

    ...
```

Réalisation du serveur

Java RMI
Exemple : Serveur

fichier HelloServeur.java

```
   ...

  public static void main(String args[]) {
        try {
              // Crée une instance de l 'objet serveur.
              HelloServeur obj = new HelloServeur("HelloServeur");
              // Enregistre l'objet créer auprès du serveur de noms.
              Naming.rebind("//suldrun/mon_serveur", obj);
              System.out.println("HelloServer" + " bound in registry");
        } catch (Exception exc) {... }
     }
}
```

Réalisation du serveur (suite)

ATTENTION : dans cet exemple le serveur de nom doit être activé avant la création du serveur

Java RMI
Activation du serveur de nom par le serveur

fichier HelloServeur.java

```
public static void main(String args[]) {
  int port;   String URL;

  try {                    // transformation d 'une chaîne de caractères en entier
    Integer I = new Integer(args[0]);    port = I.intValue();
  } catch (Exception ex) {
   System.out.println(" Please enter: Server <port>");    return;
  }

  try {
    // Création du serveur de nom - rmiregistry
    Registry registry = LocateRegistry.createRegistry(port);

    // Création d 'une instance de l 'objet serveur
    HelloServeur obj = new HelloServeur("Coucou, je suis le serveur de port : "+port);

    // Calcul de l ' 'URL du serveur
    URL = "//"+InetAddress.getLocalHost().getHostName()+":"+port+"/mon_serveur";
    Naming.rebind(URL, obj);
  } catch (Exception exc) { ...
```

Réali-sation du serveur (autre approche)

Java RMI
Exemple : Client

fichier HelloClient.java

```
import java.rmi.*;

public class HelloClient {
  public static void main(String args[]) {
   try {
     // Récupération d'un stub sur l'objet serveur.
     Hello obj = (Hello) Naming.lookup("//suldrun/mon_serveur");
     // Appel d'une méthode sur l'objet distant.
     String msg = obj.sayHello();
     // Impression du message.
     System.out.println(msg);
   } catch (Exception exc) { ... }
  }
}
```

Réalisation du client

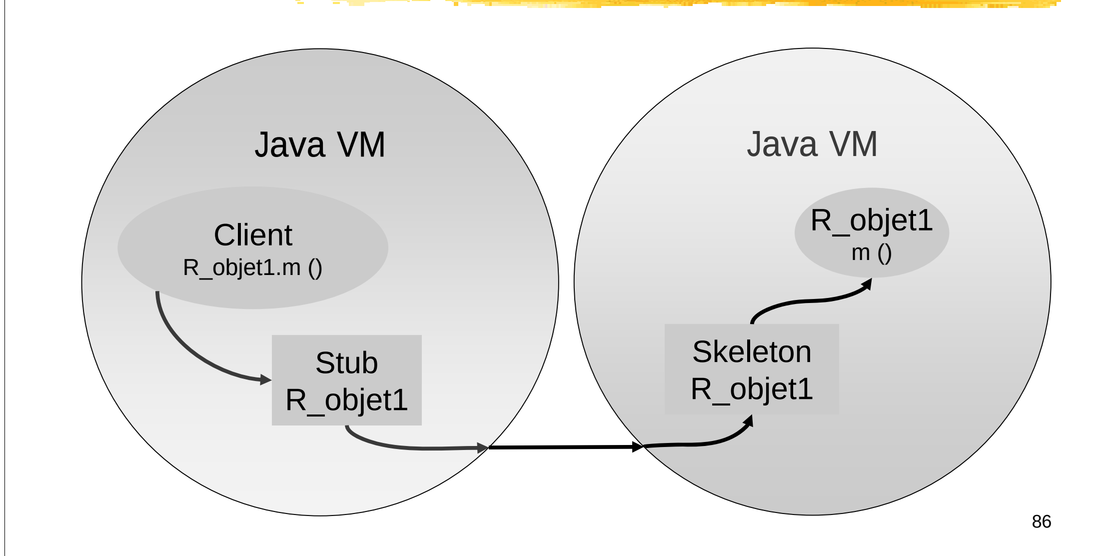
Java RMI
Compilation

- Compilation de l 'interface, du serveur et du client
  - javac Hello.java HelloServeur.java HelloClient.java
- Génération des talons
  - rmic HelloServeur
    - skeleton dans HelloServeur_Skel.class
    - stub dans HelloServeur_Stub.class.

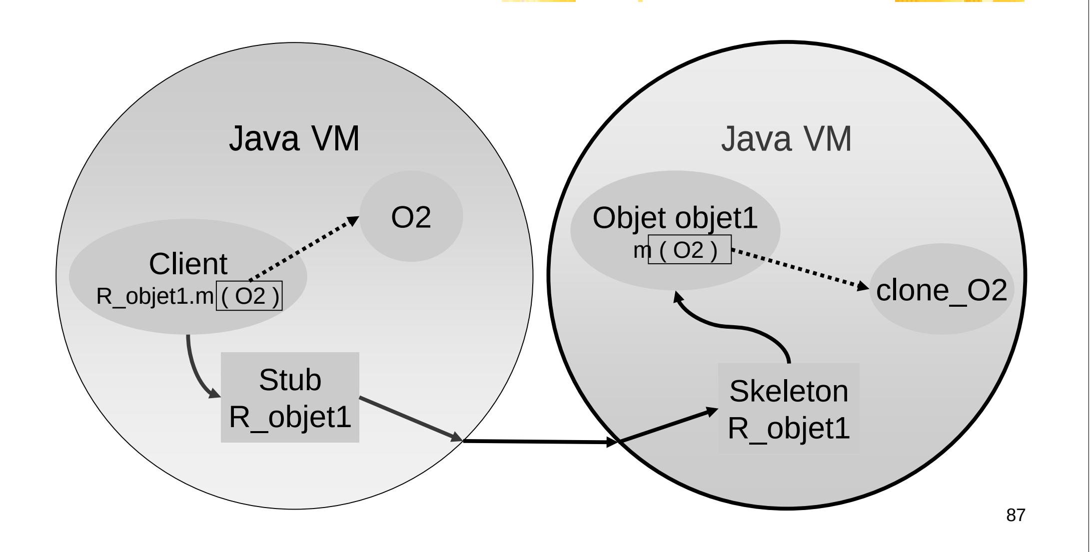
Java RMI
Déploiement

- 1) Activation du serveur de nom
  - start rmiregistry (W95) ou rmiregistry & (Unix)
- 2) Activation du serveur
  - java HelloImpl
  - java -Djava.rmi.server.codebase=http://suldrun/...
    - path indiquant à quelle endroit la machine virtuelle cliente va pouvoir chercher le code du stub
    - Nécessaire si le client et le serveur ne sont pas sur la même station
- 3) Activation du client
  - java HelloClient

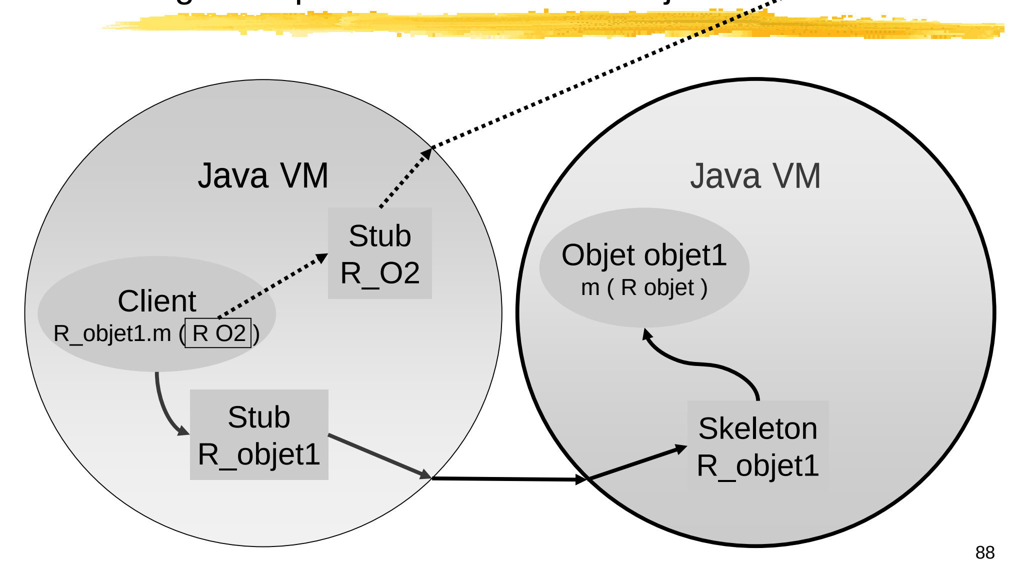
Java RMI
Principe de l 'appel de procédure



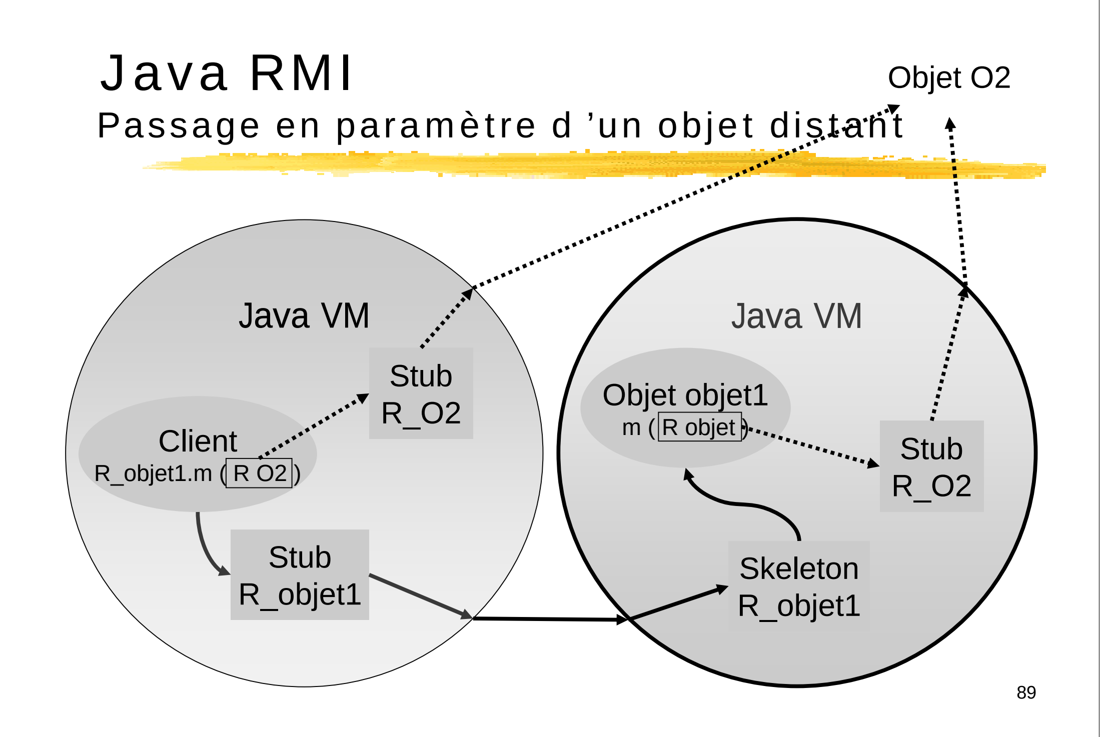
Java RMI
Passage en paramètre d 'un objet local



Java RMI
Passage en paramètre d 'un objet distant



Java RMI
Passage en paramètre d 'un objet distant



Chargement dynamique et sécurité

- Si le code du stub n 'est pas présent sur le site local, le protocole RMI prévoit le chargement dynamique du stub en utilisant un serveur web et le protocole HTTP
  - java -Djava.rmi.server.codebase=http://suldrun/...
- Si chargement dynamique...
  - le chargeur dynamique utilisé par RMI (RMIClassLoader) regarde si la classe demandée correspond au niveau de sécurité requis
    - utilisation d 'un SecurityManager
  - créer et installer le « gestionnaire de sécurité »
    - System.setSecurityManager(new RMISecurityManager());

Java RMI : bilan

- Très bon exemple de RPC
  - facilité d 'utilisation
  - intégration au langage Java et à l 'internet
  - utilisation de l 'apport de Java
    - Hétérogénïté des plateformes -> machine virtuelle
    - Passage par valeur -> sérialisation
    - Persistance -> sérialisation
    - Absence de talon -> chargement dynamique
    - Désignation -> URL
- Des ouvertures
  - Serveurs multiplexés ?
  - Sémantique en cas de panne ?
  - Utilisation d 'autres protocoles réseaux ?

RPC
Autres mises en œuvre

- Migration de code
  - code et données de la procédure distante sont amenés sur le site appelant pour y être exécutés en appel normal.
  - Avantages Inconvénients:
    - univers de systèmes homogènes.
    - faible volume de codes et de données globales
    - exclusion mutuelle entre les exécutions.

RPC
Autres mises en œuvre

- Mémoire virtuelle répartie
  - Appel distant réalisé au dessus d'une couche de mémoire virtuelle répartie. L'appel se fait en mémoire comme si la procédure était locale
    - Analogie avec une stratégie de chargement de pages à la demande : défaut de page sur le début du code de la procédure
  - Avantages Inconvénients
    - univers de systèmes homogènes.
    - Permet les gros volume de codes ou données peu visité
    - usage important de pointeurs.
    - peu de problèmes d'entrelacement sur les données globales.

Performances des RPCs
[Schroeder & Burrows 90]

	Microseconde
Client (appel)	
Programme appelant (boucle pour appel répétitif)	16
Talon client (appel & retour)	90
Initialisation de la connexion	128
Envoie du paquet d'appel	27
Serveur	
Réception du message	158
Talon serveur (appel & retour)	68
Exécution d'un service nul (appel & retour)	10
Envoie des résultats	27
Client (réception des résultats)	
Réception des résultats	49
Terminaison	33
Total	606

Performances des RPCs
[Schroeder & Burrows 90]

	Microsecondes
Null ()	
Appel serveur, stub et RPC runtime	606
Envoi/réception du paquet d'appel (74 octets)	954
Envoi/réception du paquet retour (74 octets)	954
Total	**2514**
MaxResult (b)	
Appel serveur, stub et RPC runtime	606
Marshall 1440 octets pour le résultat	550
Envoi/réception du paquet d'appel (74 octets)	954
Envoi/réception du paquet retour (1514 octets)	4414
Total	**6524**

Facteurs d'amélioration
[Schroeder & Burrows 90]

	0 octets	1440 octets
Vitesse processeurs X 3	52 %	36 %
Attente active	17 %	7 %
Contrôleur amélioré	11 %	28 %
Protocole transport en assembleur	10 %	4 %
Protocole amélioré	8 %	3 %
Pas de contrôler d'erreur	7 %	16 %
Vitesse réseau X 10	4 %	18 %
Suppression des couches IP/UDP	4 %	1 %

Quelques travaux complémentaires

- Tendances
  - Parallélisation, flots
  - Tolérance aux défaillances
  - Intégration dans les langages de haut niveau
  - Amélioration des performances brutes
  - Usage local (appel entre contexte)
- Appels parallèles
  - [Satyanarayanan & Siegel 90]
- Flots
  - Mercury [Liskov & Shrira 88]
- RPC "léger"
  - [Bershad & al. 90]

RPC à sens unique

- Envoie d 'un message asynchrone pour déclencher une procédure
  - la procédure ne doit pas avoir à retourner des résultats
  - récupération d 'une réponse par un mécanisme similaire
    - invocation d 'actions du client par le serveur
    - le client doit avoir prévu les actions correspondant aux différents types de réponses
  - pas d 'information sur la terminaison du travail demandé
    - mais envoie d 'un message sous la forme syntaxique d 'un appel de procédure (qui ne doit pas avoir de résultat à rendre)
  - c 'est le modèle des langages acteurs
    - ABCL, ACTALK, HYBRID

RPC asynchrone

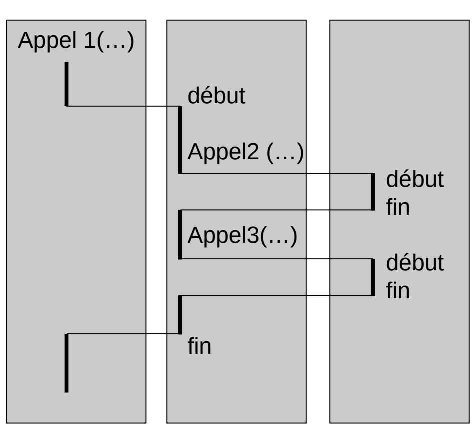
- Le client poursuit son exécution après l 'émission du message d 'appel
  - la procédure distante s 'exécute en parallèle avec la poursuite du client et retourne les paramètres résultats en fin de son exécution
  - le client récupère les résultats quand il en a besoin (primitive spéciale)
  - avantage : parallélisme plus important
  - critique le client ne retrouve pas la sémantique de l 'appel de procédure
    - contrôle de la récupération des résultats : pb de synchronisation (risque d 'erreur)

Appel asynchrone avec futur

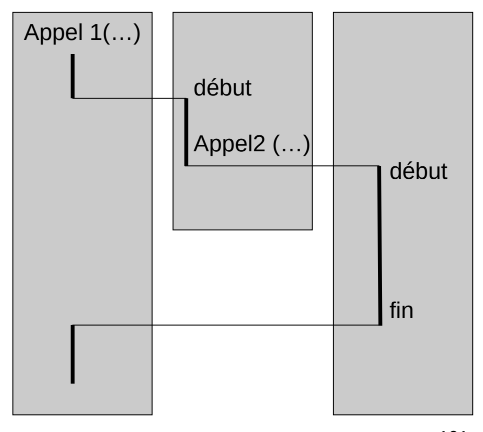
- Futur
  - objet particulier pour la récupération des résultats
  - futur explicite :
    - construction avant l 'appel de l 'objet dans lequel les résultats seront déposé
  - futur implicite :
    - c 'est le mécanisme d 'appel qui construit les objets résultats
  - mode d 'utilisation :
    - la lecture rend un résultat nul si le résultat n 'est pas disponible
    - la lecture bloque le client si le résultat n 'est pas disponible

Appel de procédure
Schéma d 'exécution pouvant être déduit

- Appels de procédure imbriqués



- Schéma à continuation



Les limites du modèle client-serveur

- modèle de structuration
  - permet de décrire l'interaction entre deux composants logiciels
    - absence de vision globale de l'application
  - schéma d'exécution répartie élémentaire (appel synchrone)
    - absence de propriétés portant sur la synchronisation, la protection, la tolérance aux pannes, . .

Les limites du modèle client-serveur

- services pour la construction d'applications réparties
  - le RPC est un mécanisme de “bas niveau”
  - des services additionnels sont nécessaires pour la construction d'applications réparties (désignation, fichiers répartis, sécurité, etc.)
    - CORBA, EJB, ...
- outils de développement
  - limités à la génération automatique des talons
  - peu (ou pas) d'outils pour le déploiement et la mise au point d'applications réparties

Construction d 'applications réparties à l 'aide des RPC

- ATTENTION : client-serveur (et objets répartis) masquent la différence entre appel de procédure et appel de procédure à distance
  - une telle application présente toutes le caractéristiques d 'une application répartie
    - conception (et intégration de logiciel existant)
    - désignation (et protection)
    - synchronisation (et contrôle de la concurrence)
    - tolérance aux pannes (et disponibilité des services)
    - performances (et équilibrage de charge)
    - disponibilité d 'outils conviviaux pour la conception, le déploiement et la mise au point)

Modèle client-serveur Bibliographie

- A.D. Birrell and B.J. Nelson (Papier de référence)
  - "Implementing remote procedure calls"
    ACM Trans. on Comp. Syst., vol. 2(1), pp. 39-59, February 1984
- M.D Schroeder and M. Burrows (Mesures)
  - "Performance of Firefly RPC"
    ACM Trans. on Comp. Sys., vol. 8(1), pp. 1-17, January 1990
- B. Liskov and L. Shrira (RPC asynchrone)
  - "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems"
    Proc. of SIGPLAN, pp. 260-267, 1988
- B.N. Bershad, T.E. Anderson, E.D. Lazowska and H.M. Levy (RPC « léger »)
  - "Leightweight remote procedure call"
    ACM Trans. on Comp. Sys., vol. 8(1), pp. 37-55, January 1990
- Satyanarayanan & Siegel (Appels parallèles)
  - "Parallel Communication in a Large Distributed Environment"
    ACM Trans. on Comp., vol. 39(3), pp. 328-348, March 1990